マルチプロセッサで実現した H.264 ビデオ・デコーダ

――コンフィギャラブル・プロセッサのユーザ定義命令とオンチップ・バスを活用

東原朋成

ここでは，マルチプロセッサで実現したH.264 ビデオ・デコーダLSI の設計事例を紹介する．二つのプロセッサ・コアを使用し，ソフトウェア・デコードを行っている．ヘテロジニアスな機能分割型で，コンフィギャラブル・プロセッサ・コアを用いている．要求性能を実現するために，ユーザ拡張命令や独自のオンチップ・バスを活用している．（編集部）

2005 年11 月に開催された「マイクロプロセッサ フォーラム ジャパン 2005」において，筆者らはマルチプロセッサ構成のH.264 ビデオ・デコーダについて発表しました．本稿では，そのビデオ・デコーダをもとに，マルチプロセッサ構成（マルチコア）のシステムLSI をどのように設計していくかに焦点を当てて解説します．

1. マルチプロセッサ・コアASIC 時代の到来

LSI チップ上にマイクロプロセッサを搭載することは，新しい考えかたではありません．CMOS プロセスが進化し，ハード・ワイヤード論理のチップとそれを制御するプロセッサが一つのLSI チップに集約されるようになりました．最近では，DSP コアとCPU コアを搭載したLSI が携帯電話に組み込まれています．

● 機能分割とデータ分割

マルチプロセッサ構成のシステムLSI 設計において，プロセッサの使いかたからアーキテクチャを見ると，

- 機能分割
- データ分割
- 機能，データ分割

の3 通りが考えられます．

機能分割とは，**図1** に示すように，設計するシステム仕様を機能的に切り分け，各プロセッサに割り振る方法です．携帯電話向けなどのシステムLSI の多くは，機能分割タイプに相当します．例えば，システムを制御するプロセッサとデータを処理するDSP を，一つのLSI にまとめたような構成です．

データ分割とは，データのあるかたまりを複数のプロセッサで同時に処理することで，処理能力を高めるというよ

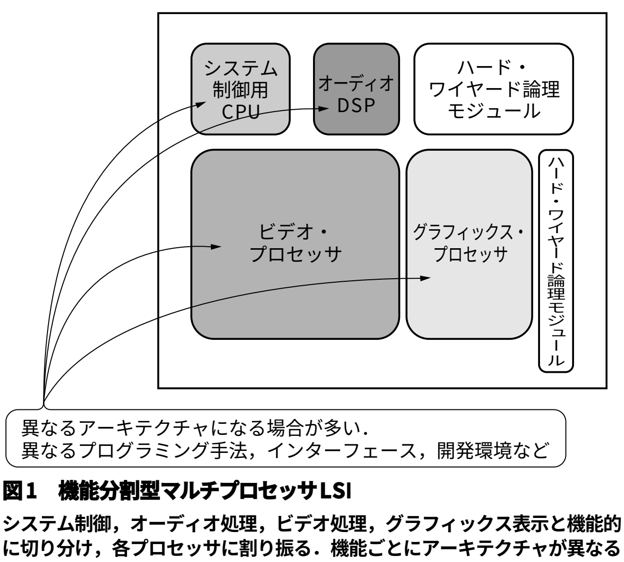
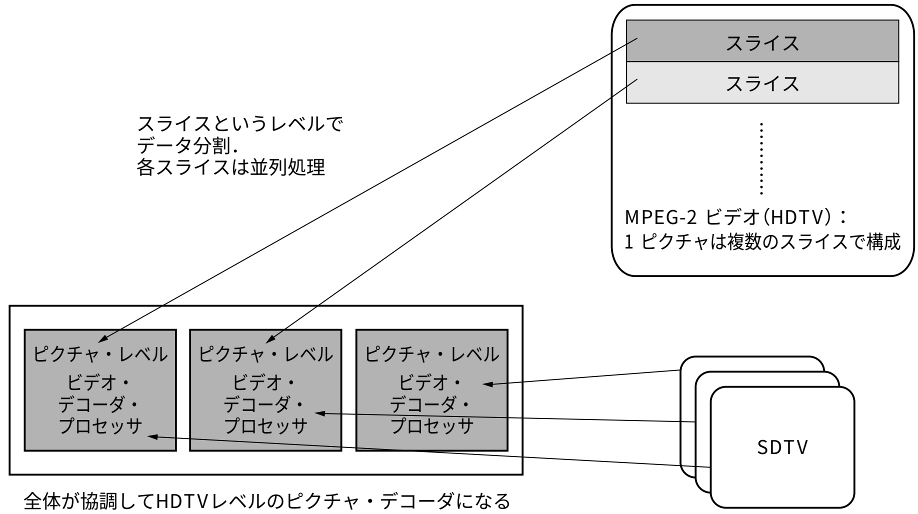


図1　機能分割型マルチプロセッサLSI

システム制御，オーディオ処理，ビデオ処理，グラフィックス表示と機能的に切り分け，各プロセッサに割り振る．機能ごとにアーキテクチャが異なるプロセッサが使われる場合が多い．

KeyWord　マルチプロセッサ，H.264，機能分割，データ分割，ホモジニアス，ヘテロジニアス，オンチップ，ネットワーク，コンフィギャラブル・プロセッサ，Xtensa

図2
データ分割型マルチプロセッサLSI
高精細テレビ（HDTV）デコーダの例を示す．1本のHDTVビット・ストリーム，もしくは3本の標準テレビ（SDTV）ビット・ストリームを同時にデコード可能な構成である．3個のプロセッサをそれぞれ独立したMPEGデコーダとして設計した．HDTVビット・ストリームでは，一つのプロセッサをマスタとし，スライス・レベルのデータを各プロセッサのMPEGデコーダで処理させる．SDTVビット・ストリームは一つのプロセッサで処理できるので，本LSIでは三つのビット・ストリームを同時にデコードできる．



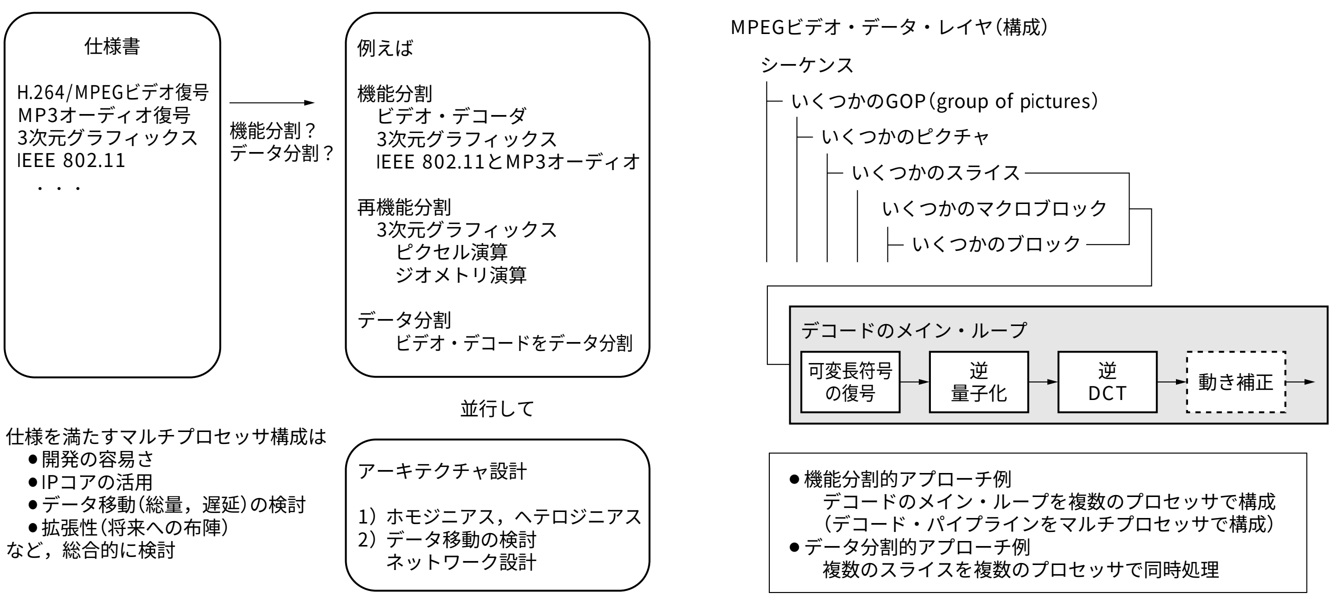


図3　マルチプロセッサLSI設計の検討点
HDTVデコーダを例に，検討点を示している．ピクチャ，GOP，シーケンスの各レベルは，デコード処理上クリティカルではないので除外している．

うな方法です．**図2**には，高精細テレビ（HDTV：high definition television）のMPEG-2ビデオ・デコーダの例を示しています．ここでは，スライスというデータ・セットを三つのプロセッサに割り振っています．もちろん，HDTVのMPEG-2ビデオ・デコーダを処理フローに従って機能分割することも可能です．実際には，**図3**に示すように，システム仕様によって適切な構成を見つける必要があります．

● ホモジニアスとヘテロジニアス

マルチプロセッサ・システムを別の観点から眺めると，
- 同一プロセッサ・コアによる構成（ホモジニアス）
- 異なるアーキテクチャのプロセッサ・コアによる構成（ヘテロジニアス）

の二つのアーキテクチャがあります．

ホモジニアスなマルチプロセッサには，タイルのようにプロセッサ・コアを並べて，同一プログラムを実行するネットワーク・プロセッサの設計例などがあります．タイプ的にはデータ分割というケースが多く，複数のデータを同時処理することで処理能力を上げています．

ヘテロジニアスな構成には，携帯電話に搭載されるCPUとDSPのコンボ・チップなどがあります（**図1**を参照）．ヘテロジニアスなシステムは，解決しようとする問題に最適

コラム　マルチプロセッサ・コアLSI開発の憂うつ

マルチプロセッサ・コアLSIの開発でもっともたいへんなことは，ソフトウェアの開発だと思います．ただでさえソフトウェア開発者や検証エンジニアの不足が叫ばれ続けています．それすら解決できず，アウトソーシングでなんとかしのいでいるといった状況において，格段に複雑になるマルチプロセッサ・ベースのソフトウェア開発をこれから進めなくてはなりません．

マルチプロセッサ・コアのソフトウェア開発をアウトソーシングできるのかという問題もあります．アウトソーシングをしようにも，技術的，経験的に信頼のおける会社があるのか，アウトソーシング先の実力を測れるエンジニアが社内にいるのかなど，いくつもの問題が存在するからです．

マルチプロセッサのソフトウェア開発は容易ではありません．いくつもの異なるアーキテクチャを同時に扱わなければならないこともあります．設計支援ツールの開発が進んでいますが，異なるアーキテクチャのプロセッサを統合的に扱えるツールとなると限られてきます．実機デバッグも欠かせませんが，チップ内に収まったプロセッサですから従来のICEを用いることはできません．JTAGポートなどを用いたデバッガが開発されていますが，異なるコアをうまくデバッガにインターフェースすることは，実際には容易ではありません．

なアーキテクチャのプロセッサを用います．しかし，プロセッサ間のインターフェースや開発ツールの問題などを考えると，ホモジニアスな構成と比較して複雑になる傾向があります（上掲のコラム「マルチプロセッサ・コアLSI開発の憂うつ」を参照）．

ホモジニアスとヘテロジニアスの両方からなるマルチプロセッサ・システムも存在します．例えばハイエンドのゲーム用グラフィックス・チップでは，ピクセル・レベルの演算にホモジニアスなマルチプロセッサ構成をとりつつ，それにサラウンド・サウンドなどの高度なオーディオ・プロセッサを付加したチップが開発されています．

● オンチップ・ネットワークが難しい

LSI製造技術や開発ツールの進化により，大量のゲート（トランジスタ）を搭載したLSIを開発できるようになりました[注1]．

一つのLSIに，プロセッサ・コアを数個はもちろんのこと，数百個搭載することすら“可能”です．可能ではあるのですが，詰め込んだというだけでは，実はどうにもなりません．その理由は，チップ上のネットワークです．

ネットワークとは，データや命令（コマンド）をやりとりする径路を指します．プロセッサが外部（例えばほかのプロセッサや外部メモリ，プロセッサ外のハードウェア資源など）と協調して何かを実行しようとする場合，データとコマンドのやりとりは欠かせません．そのための径路（ネットワーク）がチップ上に必要になります（**図4**）．

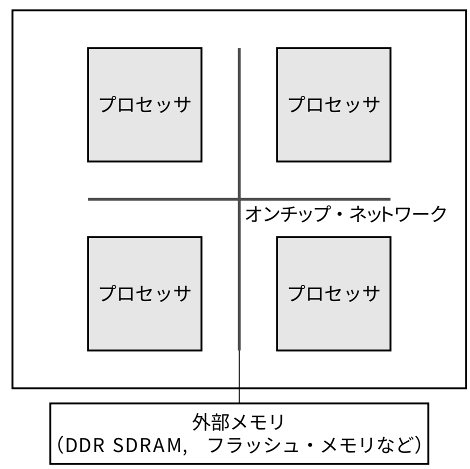


図4　オンチップ・ネットワーク
ネットワークとしては，従来，バス構造がよく使われてきた．ほかに，階層バスやリング構造（Cellなど），メッシュ構造（ネットワーク・プロセッサなど），ツリー構造，スター型などもある．究極の解はなく，アーキテクトの腕しだいというのが現状である．

ネットワークとしては，従来，バス構造がよく使われてきました．ARMコアのAMBAバスなどがよく知られています．

バス構造では，複数のブロックからのバス・アクセスを調停しなければなりません．そこでアクセス遅延の問題が生じます．調停によってバス・アクセスが待たされると，ハードウェアのクリティカルな性能制約を満たさない，ソフトウェアの実行が遅くなるといった問題に直面します．また，プロセッサ間でデータを共有することを考えると，バス上のデータ転送量が増大します（**図5**）．

データ共有には別の側面の問題も生じます．キャッシュ・メモリを搭載するプロセッサで起こるデータの一貫性の問題です．

鉄道を例に，ネットワークを考察します．鉄道サービスでは，需要の増大などに対応するために，まずはスケジュ

注1：可能とは言っても，コストや消費電力など，技術的，ビジネス的に解決しなくてはならない課題は山ほどある．

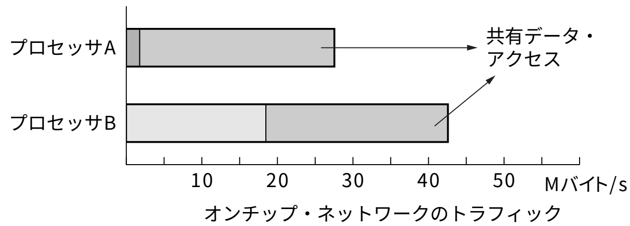


図5　データ・バスのオーバヘッド
複数のプロセッサ間でデータを共有すると，ネットワーク・アクセスが増大する．また，外部メモリなどのアクセスも発生する．したがって，バスのバンド幅やレイテンシが性能を左右することになる．

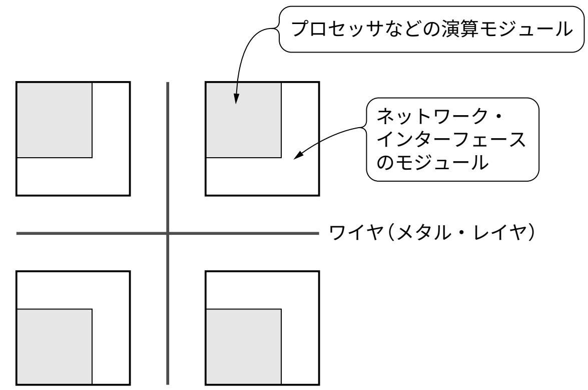


図6　ネットワークが大規模になりすぎる
データ，コマンドを移動するためのネットワークが，データやコマンドを処理するモジュールより複雑かつ大規模になる．

ール(ダイヤ)を見直し，快速や急行電車の増発の可能性を検討します．合間合間に普通電車をうまく配置することで輸送量の増大と遅延(目的地までの時間)を最適化しようとします．快速電車を増発させすぎると，普通電車の遅れがひどくなり，乗客のニーズを満たさなくなるので，おのずと制限があります．また，車両のくふうにより(例えば座席を減らす)さらなる輸送量の増大を図ろうとします．オンチップ・ネットワークの観点から考えると，これはアービトレーションのくふう，バースト・モードの導入など，ハードウェアの規模にあまり影響を与えないでデータ転送処理能力を上げようとすることに相当します．

鉄道輸送量のさらなる増大には，複々線化，並行して走るバイパス路線の導入といった対策があります．同様のことがオンチップ・ネットワーク設計においても検討されます．遅延の問題は例えばう回路になっていた駅間を接続するなどの新たな路線を設けるといった対策があります．これは例えば，メッシュ構造のネットワークを設計することに相当します．

さて，対策をたてるうえで一つ重要になるのが，ターミナル駅や大きな乗り継ぎ駅です．ターミナル駅の許容量を超える輸送は不可能です．乗降客の流れまで検討したきめの細かい最適化検討が，ターミナル駅や乗り継ぎ駅には必要になります．システムLSIにおけるもっとも重要なターミナルは，外部メモリになります．このターミナル・ポイントにおけるデータ・スループットを制御することも，オンチップ・ネットワーク設計では重要になります．

このように遅延や増大するデータ量を制御するために，複雑なネットワーク構造をエンジニアは設計してきました．外部メモリへのインターフェースを含めて，個々のアプリケーション(システムLSI)ごとに階層構造やメッシュ/リング型ネットワークとスイッチ，バッファ，アービタを手作りしてきたわけです．

IPコア・ベンダも，例えば英国ARM社はAMBA3 AXIといった従来のAMBAバスと比較して格段に性能が向上したインターフェース仕様を作成していますし，米国Sonics社のようにオンチップ・ネットワーク技術を売り物にしている会社もあります．

ネットワークがいかに重要であるかということは，最近話題の「ネットワーク・オン・チップ」ということばからもわかります．2005年11月にはフィンランドでネットワーク・オン・チップに関する国際ワークショップが開催されました．もっともこのワークショップはアカデミックな立場からの発表が比較的多く，データ，コマンドを移動するためのネットワークがデータ，コマンドを処理するモジュールより複雑かつ大規模になってしまいそうな，筆者の目からは本末転倒ではないかと思われる発表もありました(**図6**)．

筆者は，ネットワークをいかに現実的かつ効率良く設計できるかいなかで，システムLSIアーキテクトの力量が問われるのだろうと思っています．

2. H.264ビデオ・デコーダ・システムの設計

マルチプロセッサ構成のビデオ・デコーダLSIの開発事例を紹介します．このデコーダ・システムはH.264のみならず，MPEG-2，MPEG-4，VC-1といったほかの規格もサポートします．今回はH.264に絞って解説しますが，H.264設計時に定義したユーザ拡張命令は，ほかの規格もサポートできるように設計されています．機能的に固定したハー

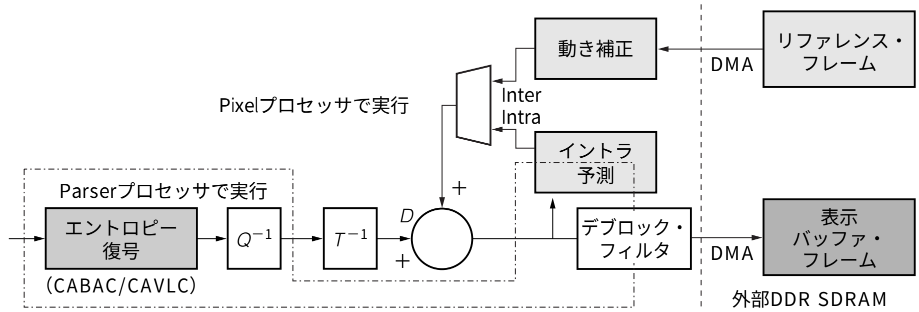


図7
H.264デコードの処理フロー
ParserとPixelの二つのプロセッサ間のデータ移動ではDMAを実行する．同期は共有データのポーリングや割り込みなどで実現．

ドウェアではなく，ゲートの有効利用ができているわけです．また，ソフトウェア・ベースの設計なので仕様の追加や変更に柔軟に対応します．H.264はビデオ圧縮の新しい規格です．ここでは規格の概要については触れませんので，参考文献(3)を参照してください．

● ソフトウェア・デコードを目ざす

図7に伸張(デコード)の処理フローを示します．ハード・ワイヤード論理のブロックを用いない完全ソフトウェア・デコードの設計を目ざしました．H.264の伸張フローは，従来の規格(MPEG-2など)と比較して複雑になっています．1プロセッサによる設計となると，非常に高い性能(Pentium級)が必要になります．組み込み向けLSIにおいては，これまで多くのハードウェア・ブロックを用いるハードウェア・ソフトウェア複合設計が主流です．ソフトウェアは制御のみをつかさどり，データ処理の大半はハードウェアで行うといったケースが多く見られました．

今回筆者らが設計したデコーダは，データの出し入れはソフトウェア制御されたハードウェア(DMA)で実行するが，データ処理はすべてソフトウェアで実行するというシステムです．ハード・ワイヤード論理を極力削減したので，

- ハードウェアの設計検証は最小
- ゲートの再利用が可能
  (固定した機能のハードウェア・モジュールが最小)
- 仕様変更，仕様追加に柔軟に対応できる
- 組み込んで検証を免れたハードウェアのバグの出現確率が低下する

などの利点があります．

ゲートを再利用でき，かつ仕様の変更や追加に柔軟に対応できることは，ソフトウェアの利点です．ソフトウェア処理の弱点としては，ソフトウェアの並列度とハードウェアの並列度の決定的な違い，命令メモリの必要性などがあります．ハードウェアは究極のパラレル・マシンであるため，潜在的バグの検証の難しさがあります．また，機能的に固定しているのでゲートを有効利用しているとは言えず，命令メモリを必要とするソフトウェアと比較して得失の比較は難しくなります．

● ヘテロジニアスな機能分割型を採用

図8が，筆者らの開発したデコーダ・システムです．二つのプロセッサ・コアを用いたヘテロジニアスなマルチプロセッサ・システムになっています．

- Parser：デコード・フローの前半部を担当．また，データの出し入れをつかさどるDMAエンジンの制御を担当．
- Pixel：デコード・フローの後半部を担当．後半部は制御系の処理は少なく，ピクセル・レベルの演算が中心．
- DMA：データの出し入れを担当．
- PIF，TIEポート，TIEキュー：プロセッサ，DMAを

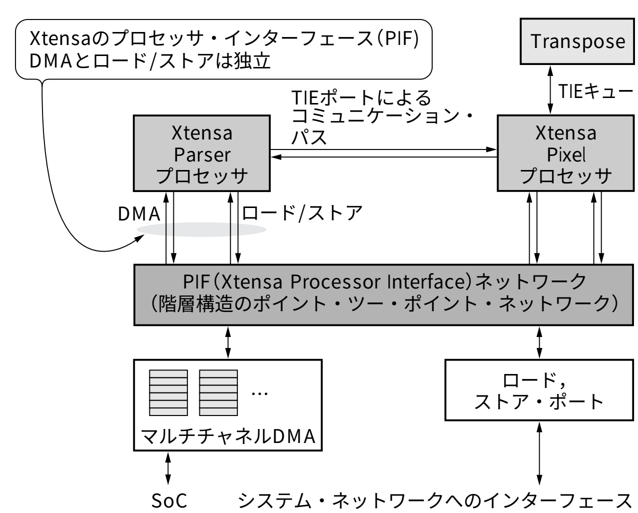


図8　設計したH.264ビデオ・デコーダ
機能分割型ヘテロジニアス構成である．一つのコンフィギャラブル・プロセッサを異なるコンフィグレーションで使用しているので，共通のツールやフローで開発できた．

接続するネットワーク．PIFはプロセッサ・インターフェース．PIFネットワークのほかにプロセッサ間を直接TIEポートでインターフェースし，ネットワーク遅延の影響を受けることなく割り込み処理などを伝える．

設計したデコーダは，機能分割型のアプローチをとりました．H.264規格を分析すると，MPEG-2とはパケット（構文上の区切り）がずいぶん異なることがわかります．MPEG-2はスライス・レイヤからくる制限がありました．つまり1ピクチャは複数のスライスで構成され，スライス・レイヤ・レベルでデータ並列処理が可能でした．しかし，H.264にはこうした制限がなく，きれいなデータ・レベルの並列処理ポイントを発見できませんでした．そこで筆者らは，デコード・フローを機能分割することにしたのです．

● ワークロードの割り当てが重要

マルチプロセッサ・システムを設計する際に重要なポイントは，

- ワークロード（作業負荷）の適切な割り当て
- データ構造，とくに共有データの取り扱い
- モジュール間同期

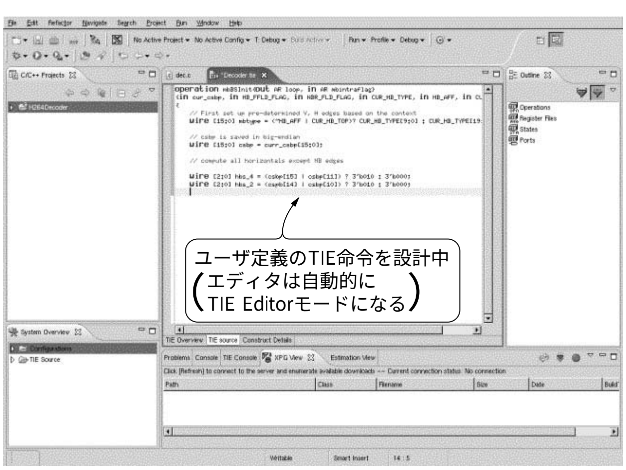
があります．

プロセッサが固定される場合，ワークロードの割り当ては，ことのほか重要かつ複雑な問題になります．処理性能が1命令分だけでも不足すると，システムとして動作しなくなるためです．しかし，当初の見積もりで機能分割し（アーキテクチャ設計），その後実際のソフトウェア開発に入ってプロセッサの性能不足が見つかる，という開発プロジェクトが後を絶ちません．ソフトウェアのマイクロ・レベルの最適化（アセンブリ言語による直接的な設計にまで至ることもある），ツール・レベルの最適化など，さまざまなアプローチを経た末，アーキテクチャの変更や仕様変更に至るケースも見受けられます．

筆者らは，幸いにしてコンフィギャラブル・プロセッサをベースに設計を進めたので，こうした問題を回避することができました．米国Tensilica社のXtensaプロセッサは，ユーザが構成（例えばキャッシュ・メモリの容量など）を変更したり，ユーザ拡張命令を追加できるプロセッサです（下掲のコラム「Xtensaプロセッサによる開発」を参照）．

コラム　Xtensaプロセッサによる開発

Tensilica社のXtensaは，ユーザが開発するシステムに最適化できるコンフィギャラブル・プロセッサ・コアです．DSP的にプロセッサを拡張することもできますし，Linuxなどを搭載するシステム・コントローラとしてコンフィグレーションすることもできます．最新のXtensa LXでは，VLIWライクな拡張命令（FLIX），TIEキュー，TIEポートなどをサポートしています．



図B-1　統合開発環境（IDE）Xtensa Xplorer

● 開発環境

Xtensaベースの開発では，統合開発環境（IDE）Xtensa Xplorerを使用します．開発エンジニアはTIE拡張命令の開発，アプリケーション・ソフトウェアの開発，デバッグ，実機システム（エミュレーション・ボード）を用いたデバッグなどは，IDEを用いて行います（**図B-1**）．

マルチコア・システムのモデリングには，XTMP（Xtensa Modeling Protocol）を使用します．このXTMPはSystemCベースのモデルともインターフェースできます．

筆者らが設計したビデオ・システムでは，Xtensa Xplorer上でマルチプロセッサ・モデルを開発しました．デコーダ・ソフトウェアを開発するエンジニアは，このモデル上でプログラムを実行することで動作検証（アウトプットの正しさとリアルタイム性の確認）を行い，その後，最終検証としてRTLシミュレーションを実施しました．XTMPは完全なCモデルなので，使用にあたってSystemCの知識は必要ありません．

コンフィギャラブル・プロセッサ・ベースのマルチコアLSIの開発では，複数のプロセッサ・コアはまったく異なる拡張，コンフィグレーションになるかもしれません（つまりヘテロジニアスなマルチプロ

● ユーザ拡張命令で必要な処理性能を得る

開発にあたり，筆者らは消費電力やコストの観点からハードウェアの設計目標を定めました(**図9**)．このハードウェアの設計目標の範囲内で，ハードウェアを最適に構成できるわけです．機能分割やデータ共有など，ソフトウェア・アーキテクチャを設計した後は，個々のソフトウェア開発を独立して行えます．プロセッサに必要な演算能力は，ユーザ拡張命令を用いて得ることができますし，演算に必要とする最適なメモリを構成することができます．

各プロセッサの構成を**図10**に示します．ユーザ拡張命令の例として，ここではH.264のデブロッキング・フィルタなどに必要な演算命令(Pixel Engine)と算術符号(CABAC)の復号に必要な命令を示します．

ビデオ関連のアプリケーションはピクセル・レベルの演算を多用します．米国Intel社のMMX(MultiMedia eXtension)のように，SIMDタイプの演算命令が非常に有効に機能します．もちろんXtensaでもSIMDデータ・タイプを定義し，SIMDデータを扱う演算命令を定義できます．**図11**にPixelプロセッサに追加したSIMDタイプ命令語の例を示します．データ・タイプはユーザ・プログラムで使用できます．新たに定義されたデータ・タイプはコンパイラによって認識されるため，特別なライブラリ・セットを作成する必要はありません．

- 物理的設計目標
  サイズ：$10.5mm^2$以下(0.13 μm CMOSプロセス)
  クロック周波数：200MHz以下
  (消費電力などの目標から設定)
- 機能(スペック)目標
  マルチスタンダード(H.264，MPEG-2/4，VC1)
  SDTVレベルの解像度，フレーム・レート(PAL，NTSC)
- ハード・ワイヤード論理を最小にする
  完全ソフトウェア・デコード

図9　設計目標
消費電力，コストの観点から設計目標を定めた．

● ユーザ定義のステートと命令で制御フローを高速化

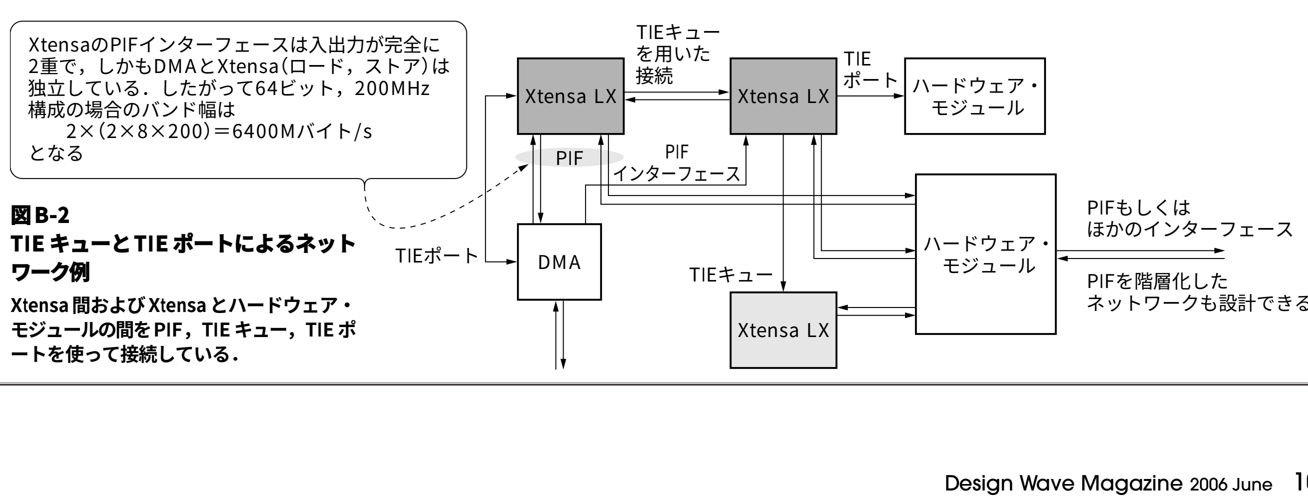
XtensaがMMXなどのマルチメディア拡張命令を備えたプロセッサと大きく異なる点の一つは，制御フローを拡張命令で高速化できることです．CABACを例に示します．**図12**には，条件分岐やループが多数見られます．演算そ

セッサ)．しかし開発ツールが統一されていれば，異なるベンダの環境を統合するといった手間はなくなります．XTMPなどを用いてマルチプロセッサを取り扱うソフトウェア・モデルを構築すれば，実機システムの挙動をソフトウェア・モデル上の挙動と比較することも可能です．このヘテロジニアスなマルチプロセッサ・システムをホモジニアスな開発環境で設計できることはXtensaプロセッサの特徴といえます．

● TIEキューとTIEポート

筆者は本誌2005年6月号において，設計者が自由に定義できる新しいタイプのインターフェース(TIEポート，TIEキュー)について解説しました[1]．TIEポートとTIEキューは，プロセッサ間，プロセッサとハード・ワイヤード論理のブロック間を直接接続でき，共有ネットワーク(バス，リング，メッシュなど)を補完するものです．共有ネットワークと1対1のネットワークを組み合わせることで，満たさなくてはならない設計条件(総データ量とデータ遅延)を現実的に解決することができます．

Xtensaのプロセッサ・インターフェース(PIF)とTIEキュー，TIEポートを活用して有機的にネットワークを構成した例を**図B-2**に示します．



図B-2
TIEキューとTIEポートによるネットワーク例
Xtensa間およびXtensaとハードウェア・モジュールの間をPIF，TIEキュー，TIEポートを使って接続している．

- Parser
  - データ・メモリ：32Kバイト
  - 命令メモリ：30Kバイト
  - データ・キャッシュ：4Kバイト
  - プロセッサ・インターフェース：64ビット
  - 64ビット命令語セット（FLIX，VLIWライクな命令語セット）
- Pixel
  - データ・メモリ：64Kバイト
  - 命令メモリ：20Kバイト
  - データ・キャッシュ：4Kバイト
  - プロセッサ・インターフェース：128ビット
  - SIMD命令語セット（8ピクセル・データを並列処理可能）

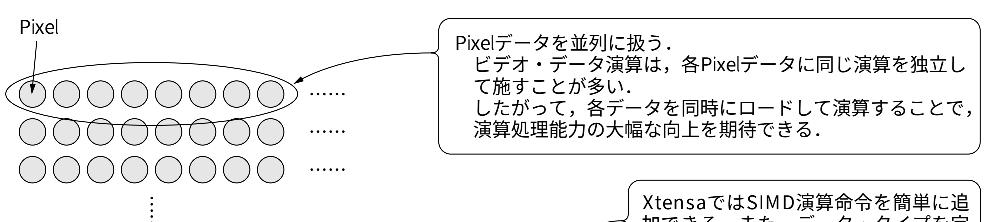
拡張命令語	種　類
cmpgt_i, cmpge_i, cmplt_i, cmple_i, cmpeq_i, cmpgtt_i, cmpget_i, cmpltt_i, cmpltt_i, cmpeqt_i, cmpeqt_i, cmpgtf_i, cmpgef_i, cmpltf_i, cmplef_i, cmpeqf_i, mvt, mvf	条件比較，条件MOV
wrnibble, rdnibble, rdbyte	ロード，ストア
ldz	
bs_loadbuffer, bs_getbits, bs_ldgetbits, bs_showbits, bs_flushbits, bs_AlignByte, bs_flushbitsstate, bs_ldz	ビット・ストリーム・ロード，ビット・ストリーム・ストア
bs_runbefore, bs_invscan, bs_ldtable8, ld_quant, bs_rdtones, bs_quant, bs_suffixsixe, bs_levcode, bs_calclevel, bs_calcindex	CAVLC
init_cabac, ld64_cntxt0_i, st64_cntxt0_i, ld64_contxt1_i, st64_cntxt1_i, ld64_cntxt2_i, st64_cntxt2_i, bs_rdlevel, st64_level_i	CABAC
bs_decisionbin, bs_bypassbin, bs_readcoeff4x4	CABAC
ldMVleft, ldMVtop, lvMVtopright, ldLeftRefFrame, ldTopRefFrame, ldTopRightRefFrame, mv_predict, getCurMVptr, getLeftMVPtr, getTopMVPtr, getTopRightMVPtr, getNeighborIndex	モーション・ベクタ予想
init_leftnnz, check_cbp_bit, calc_cntxtincr, modify_csbpnnz, read_csbp_bit, calcN	CBP
mbBsInit, mbBsLoad, mbBsGetblkIndx, mbBsMvdiff, mbBsRefdiff, MbBsCalc	Boundary Strength
ldz, calcMvXY, calcFrmXY, mvXYClip3, MvDLoad32, DMvClip3, DMvPow2Div, DMvAbs16, DMvCompUci, MvMacLLHS, MvMacLLSScale6, MvMacHLSScale8, MvMacLLSScale8	MAC，ALU，クリップ，スケール

図10
プロセッサの構成
ここではH.264のデブロッキング・フィルタなどに必要な演算命令（Pixel Engine）と算術符号（CABAC）の復号に必要な命令を示す．

	TIE命令を追加する前		TIE命令を用いて拡張		
	Rugby Cabac 5.4Mbps	Rugby Cavlc 5.4Mbps	Rugby Cabac 5.4Mbps	Rugby Cavlc 5.4Mbps	ゲート数
Parser	1,300MCPS	701MCPS	196MCPS	188MCPS	137,840
Pixel	1,218MCPS	1,217MCPS	187MCPS	185MCPS	148,711

TIE命令なしでは1.3GHzのコアが必要

TIE命令を使ったことにより200MHz以下のコアでよく，面積と消費電力を大幅に削減できた




```
ctype int8x16    // 8-bit x 16 SIMDデータ・タイプの定義
regifile vec128 128 16 //128ビット・レジスタ(16エントリ)の定義
operation simd8x16add {out vec128 o, in vec128 a, in vec128 b}{}
{
   wire [7:0] t0 = a[7:0] + b[7:0];
       ...........
  wire [7:0] t15 = a[127:120] + b[127:120];
   assign o = {t15, t14, t13, t12, t11, t10, t9, t8, t7, t6, t5, t4, t3, t2, t1, t0};
}
```

XtensaではSIMD演算命令を簡単に追加できる．また，データ・タイプを定義でき，SIMD演算命令をコンパイラが自動認識する（特別なライブラリを用意する必要はない）

SIMD型ADD命令の定義
16の8ビットADD演算を並列に実行する

図11
Pixelプロセッサに追加したSIMDタイプ命令語の例
128ビットSIMDデータ・タイプを定義した例を示す．図には示していないが，データをロード，ストアする命令をユーザが定義する必要がある．

のものは単純ですが，SIMDタイプの演算は不適格であるため，演算の並列度を上げて処理能力を高めるというMMX型のアプローチはとれません．条件が多いということは，プロセッサ・パイプラインにハザードを生じやすく，やっかいです．

Xtensaはユーザ定義のステート（状態）を設計できます．通常，プロセッサは，プログラム・カウンタをはじめ，GPR（general purpose register）のほかにいくつものステ

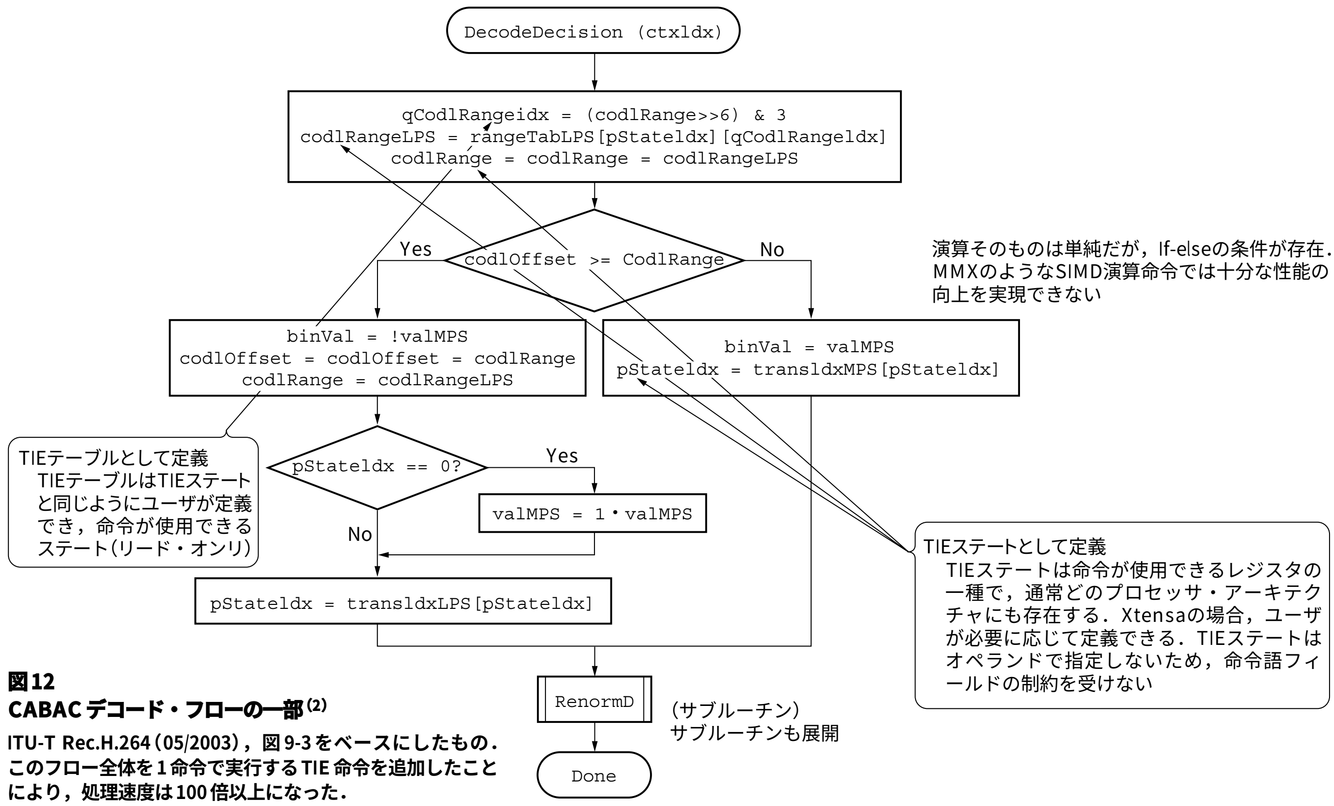


図12
CABACデコード・フローの一部(2)
ITU-T Rec.H.264(05/2003)，図9-3をベースにしたもの．このフロー全体を1命令で実行するTIE命令を追加したことにより，処理速度は100倍以上になった．

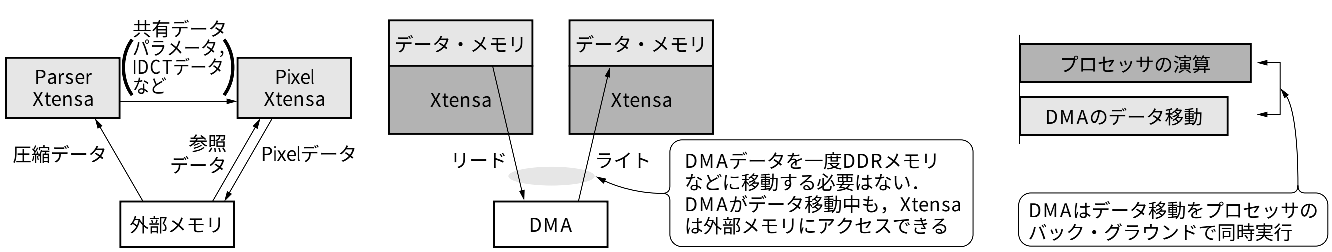


図13　デコード処理におけるデータ移動
膨大なデータをプロセッサ-メモリ間で移動，または二つのプロセッサで共有する必要がある．DMAを用いることで，外部メモリを介さずにデータ移動が行える．また演算を並列化できる．

ートを記憶しています．Xtensaでは，プロセッサが最初から持つステートのほかに，ユーザが独自のステートを定義できます．ユーザ定義のステートは，ユーザ定義の命令で参照・更新できます．ステートはダイレクトに参照されるので，GPRを参照する際に必要なオペランド・フィールドは不要です．つまり，命令語は自由にステートを使えます．もちろんステートもハードウェア・モジュールですから，むやみにステートを定義するとコスト面に影響します．

CABAC用の命令を設計するにあたり，ユーザ定義のステートを活用しました．ステートを用いてCABACフローの条件分岐とループをすべて1命令に収めたのです．ステート・マシンをソフトウェアの命令語にしたことになります．ステートはCABACに特化したものではありません．実際，筆者らの設計では同じステートはCAVLC(CABACと並ぶもう一つの圧縮手法)などを取り扱う命令語でも使用しています．

● DMAでデータを移動する

RISCプロセッサの命令語は，レジスタ上のデータのみを処理します．したがって，メモリ上のデータはロード，ストア命令でレジスタとやりとりする必要があります．また，マルチプロセッサでは，プロセッサ間でデータのやりとりが必要になります．

設計したデコーダでは，大ざっぱに**図13**に示すような

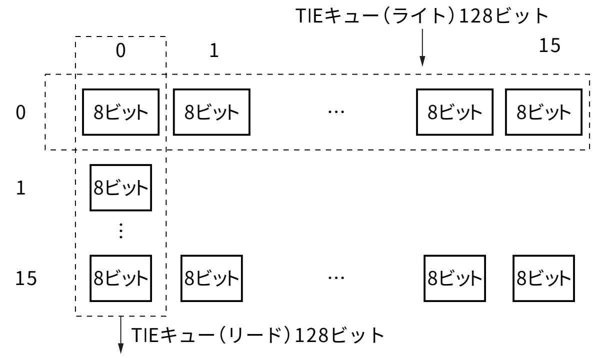


図14　転置メモリ
TIEキューに接続すると，転置に必要なサイクル数がゼロになる．

データのやりとりが必要になります．

プロセッサ間でデータを共有するとなると，キャッシュ・データの不一致の問題が生じます．それを避けるいちばん手っ取り早い手法は，共有データについてキャッシュを介さないことです．ただし，この場合，プロセッサの性能が大きく低下します(すべてのロードがキャッシュ・ミスということになる)．また，共有データを適切に同期させなくてはなりません．こうしたことをソフトウェアで処理すると，大きなオーバヘッドにつながります．デコード・フローにはまったく現れないデータの移動になるためです．そこで筆者らは，プロセッサ間，プロセッサとメイン・メモリの間のデータのやりとりをDMAで実行することにしました．

Xtensaは，データ・メモリや命令メモリを外部DMAが直接アクセスできるようなプロセッサ・インターフェースを備えています．デコード・フロー上では複数のデータがやりとりされますから，独立した複数のDMAチャネルを用意しました．個々のDMAチャネルはソフトウェアで完全に制御できます．DMAアクセスのオーバヘッドを極力最適化できるように，DMAは開始・終了をソフトウェア・ポーリング(DMAステータスをリードする)，または割り込みの二つの手法で知ることができるようになっています．

ソフトウェアでマルチバッファを構成し，データ処理とDMAによるデータ移動を並列化することで，データ移動のオーバヘッドを隠ぺいしています．

● 電力制御やデータ転置にオンチップ・バスを活用

設計したビデオ・システムでは，TIEキューとTIEポートも多用しています．二つのXtensaのうち，Parserがマスタになります．Parserが何もデータを持たない場合，PixelはIDLEになります．そこで消費電力を低減するために，ParserでPixelのクロック・ゲートを直接制御できるようにしました．XtensaにはRunStallという入力があり，外部から直接クロックをOFFできます．ParserのTIEポート(出力)をPixelのRunStallに接続することで，この機能を実現しています．また，時としてParserはPixelに割り込みをかける必要があるのですが，これもTIEポートを用いました．

Pixelには，Transposeというブロックがあります．これは特別なメモリ・アレイで，2次元行列の転置を実行するためのものです．ビデオ・データ処理では，2次元行列の演算がよく実行されます．SIMD命令が威力を発揮するわけですが，行計算を実行した後で同じ処理を列に施そうとする場合，データ・セットを転置する必要があります．転置のための命令はMMXなどでも用意されているのですが，意外と多くのサイクル数が必要です．時として演算そのものより転置にサイクル数が必要になるといった場合もあります．

設計したビデオ・デコーダでは当初，転置をサポートする拡張命令を設計していました．しかし，転置を施す処理が頻繁に出てくることから，転置をメモリ内で自動的に行える特殊なメモリ・アレイを設計しました(**図14**)．それをTIEキューに接続すると，転置に必要なサイクル数がゼロになります．このメモリは転置のみならず，将来設計予定のエンコーダ・システムにおいて別の機能を担う予定です．こうした特別なメモリ・ユニットは，通常のメモリ・インターフェース(キャッシュなど)には直接接続できませんが，TIEポート，TIEキュー・インターフェースであれば接続可能です．

参考・引用*文献

(1) 東原朋成；マルチプロセッサLSIに適したオンチップ・ネットワーク，pp.87-93，Design Wave Magazine，2005年6月号．
(2)* ITU-T Rec.H.264，May 2003．
(3) 特集2 動画像圧縮伸張規格H.264のハード&ソフト設計，Design Wave Magazine，2004年8月号．

とうはら・ともなり
米国Tensilica社